SEUPD@CLEF: Team CLOSE

Antolini Gianluca[1], Boscolo Cegion Nicola[1], Cazzaro Mirco[1], Martinelli Marco[1], Safavi Seyedreza[1] and Shami Farzad[1]

[1] *University of Padua, Italy*

Abstract

This paper presents the work of the *CLOSE* group, a team of students from the University of Padua, Italy, for the CLEF[1] LongEval LAB 2023 Task 1 [1]. Our work involved developing an Information Retrieval system that can handle changes in data over time while maintaining high performance. We first introduce the problem as stated by CLEF and then describe our system, explaining the different methodologies we implemented. We provide the results of our experiments and analyze them based on the choices we made regarding various techniques. Finally, we propose potential avenues for future improvement of our system.

Keywords

Information Retrieval, Search Engines, Longitudinal Evaluation, Model Performance

1. Introduction

Recent research has shown that the performance of information retrieval systems can deteriorate over time as the data they are trained on becomes less relevant to current search queries. This problem is particularly acute when dealing with temporal information, as web documents and user search preferences evolve over time. In this paper, we propose a solution to this problem by developing an information retrieval system that can adapt to changes in the data over time, while maintaining high performance.

Our approach involves using the training data provided by *Qwant* [2] search engine, which includes user searches and web documents in both French and English. We believe that this data will enable our system to better adapt to changes in user search behavior and the content of web documents.

The remainder of this paper is organized as follows: Section 2 describes our approach in more detail, including the different techniques we used. Section 3 explains our experimental setup, including the datasets and evaluation metrics we used. Section 4 presents our main findings and analyzes them based on the choices we made regarding various techniques. Finally, Section 5 summarizes our conclusions and outlines potential avenues for future work.

[1] Conference and Labs of the Evaluation Forum

"Search Engines", course at the master degree in "Computer Engineering", Department of Information Engineering, and at the master degree in "Data Science", Department of Mathematics "Tullio Levi-Civita", University of Padua, Italy. Academic Year 2022/2023

gianluca.antolini@studenti.unipd.it (A. Gianluca); nicola.boscolocegion.1@studenti.unipd.it (B. C. Nicola); mirco.cazzaro@studenti.unipd.it (C. Mirco); marco.martinelli.4@studenti.unipd.it (M. Marco); seyedreza.safavi@studenti.unipd.it (S. Seyedreza); farzad.shami@studenti.unipd.it (S. Farzad)

CEUR Workshop Proceedings (CEUR-WS.org)

2. Methodology

2.1. General Overview of Our IR System

In the development of our IR[1] system, we followed the traditional *Y model* suggested by Apache Lucene [3], represented in the figure below.

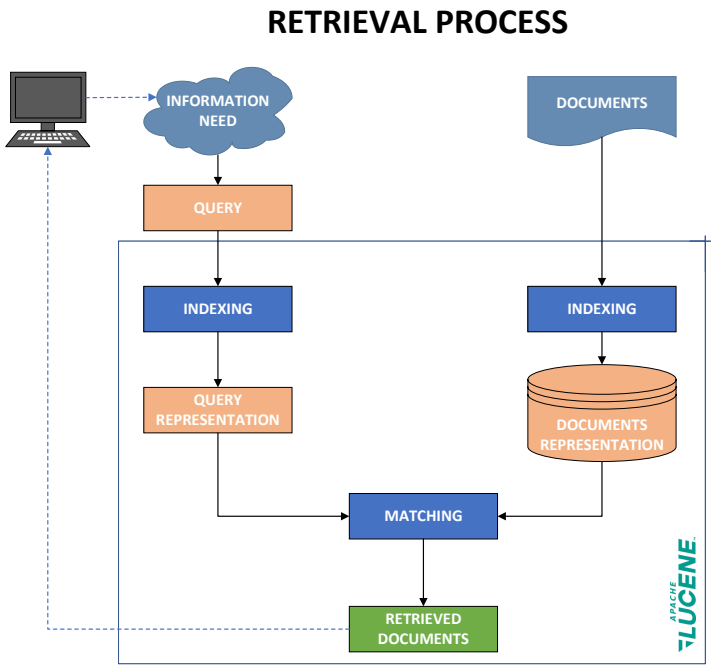

Figure 1: Apache Lucene Y Model.

The workflow of our system, starting from the train collection provided by CLEF [1], is as follows:

1. **Parsing**: The first phase consists of parsing the documents in the collection, which is a pre-processing operation performed to clean them from unnecessary noises. Since the collection is composed of web pages, the documents contain many leftovers like JavaScript scripts, HTML and CSS codes, HTTP and HTTPS URIs, and so on. The purpose of this phase is to ease the processing performed in the following phases.
2. **Indexing**: Each parsed document is then analyzed and indexed keeping only the necessary information. Indexed documents are composed of two fields: an *id* field, containing the identifier of the document in the collection, and a *content* field, containing the entire body of the document cleaned by the parsing and indexing phases.

[1]Information Retrieval

3. **Query Formulation**: Topics are then parsed using the same analyzer used for documents, and used to formulate queries. For each topic, together with the already provided query, around 15 others are computed and used altogether for searching relevant documents.
4. **Re-ranking**: The retrieved documents are re-ranked combining the scores obtained with two different approaches which compute the similarity of the document to the query given in the topic.

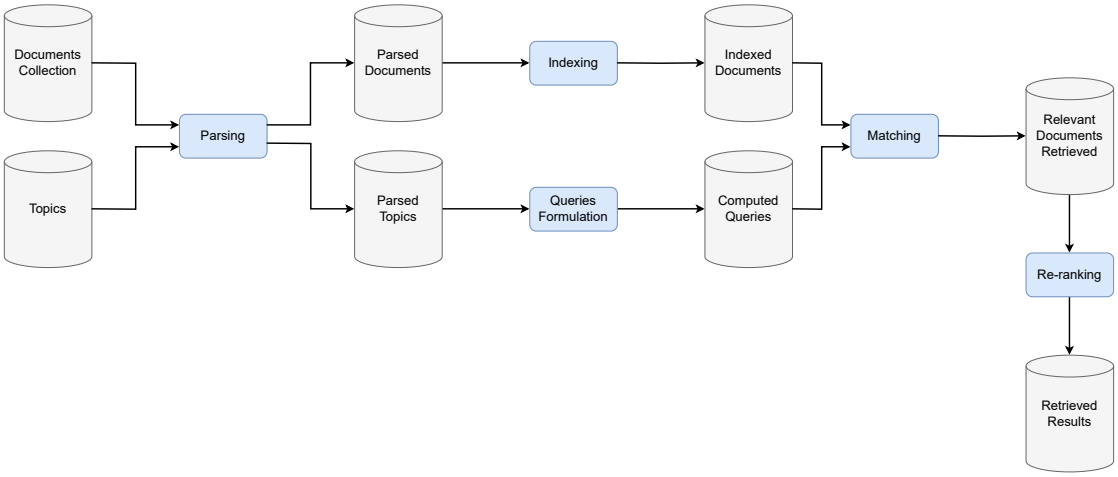

Figure 2: Workflow of the IR system implemented by *CLOSE*.

2.2. Project Structure

The project is developed mainly in *Java*, with the addition of some *Python* scripts for performing the expansion of the queries and for the re-ranking of the retrieved documents for each topic. The overall structure is as follows:

- **experiment**: This folder contains the results of the indexing obtained with different approaches, stored in different sub-folders.
- **python_scripts**: This folder contains the above-mentioned *Python* scripts, together with other *txt* and *JSON* data files produced or read by these scripts (i.e. query expansions, synonyms, etc.).
- **src/main/java**: This folder contains all the *Java* classes used for the implementation of the IR system, divided into the following packages:
  - **parser**: This package contains the classes involved in the pre-processing of the document, i.e., the parsing phase.
  - **analyzer**: This package contains the classes used for analyzing the parsed documents and topics of the collection.
  - **indexer**: This package performs the indexing of the documents processed by the analyzer.

- **searcher**: Once the documents are indexed, the queries contained in the topics of the collection are parsed and then expanded in this package. Relevant (or assumed to be so) documents are retrieved and scored.
- **utils**: This package contains general utility classes, one of which performs the re-ranking of the documents retrieved by the *searcher*.

A schema of the overall project structure is the following:

```
/code
|------ experiment
|------ python_scripts
|------ src
          |------ main
                    |------ java
                              |------ parser
                              |------ analyzer
                              |------ indexer
                              |------ searcher
                              |------ utils
```

2.3. Class Diagram

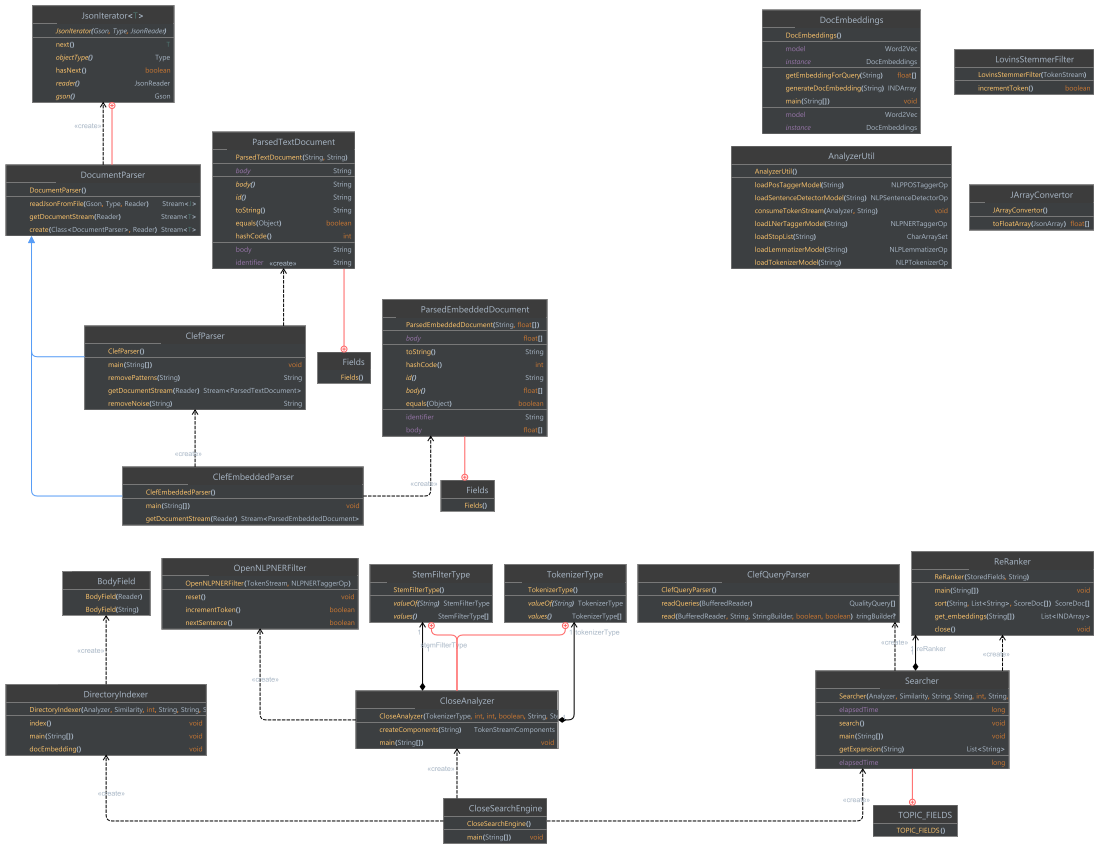

Figure 3: Diagram of the classes implemented by *CLOSE* IR system

2.4. Parser

As stated before, the documents in the collection provided by the CLEF *LongEval LAB 2023* [1] are essentially the corpus of web pages, to better represent the nature of a web test collection. From this, the need for performing a pre-processing phase of parsing the documents before analyzing and indexing them arises. In this phase, the documents are cleaned from all the residuals of codes not useful for our purposes. We first created an abstract class *DocumentParser* and then extended it by implementing a custom *ClefParser* class, which contains many functions for removing sundry types of noises that can be present in documents. This was the result of the trial and error approach we adopted for implementing this class:

- We started by reading a large statistical sample size of the documents in the collection to decide which types of noises needed to be removed.
- Then we implemented the parser and ran it.
- The results of the parsing were stored, and a sample of the parsed documents was analyzed to start this procedure again.

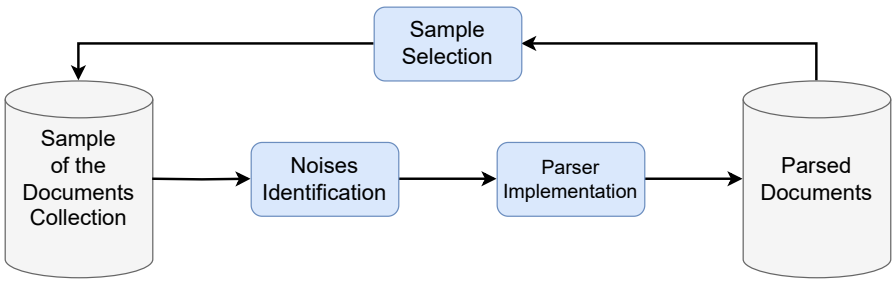

Figure 4: Workflow of the parser implementation.

The types of noises we tried to remove are the following:

- *JavaScript* scripts,
- *HTTP* and *HTTPS* URIs,
- *HTML* tags and *CSS* stylesheets,
- *XML* and *JSON* codes,
- Meta tags and document properties,
- Navigation menus,
- Advertisements,
- Footers,
- Social media handlers,
- Hashtags and mentions.

We also identified some patterns of words and symbols to remove:

- Two words separated by an underscore, like *word1_word2*
- Two words separated by a colon, like *word1:word2*
- Two words separated by a point, like "*word1.word2*"

We used *Regular Expressions* [4] to define the patterns to identify and remove. The code below shows the *removePatterns* function, used by the parser for removing the patterns mentioned above together with the *HTTP* and *HTTPS* URIs.

```
// Function to remove patterns like "word1_word2", "word1.word2",
    "word1.word2", and HTTP/HTTPS URIs from a string
public String removePatterns(String input) {

    // Define regular expression pattern to match the desired patterns
    Pattern pattern = Pattern.compile("(\\w+)[_.:](\\w+) |
        (https?://[\\w-]+(\\.[\\w-]+)+([\\w.,@?^=%&:/~+#-]*[\\w@?^=%&/~+#-])?)
        ");

    // Replace all matched patterns with a space character
    Matcher matcher = pattern.matcher(input);
    String output = matcher.replaceAll(" ");

    return output;
}
```

After different tries, we decided to use only the function above for removing noises, which gave an improving in MAP[2] of +0.5%, together with another one used to remove *JavaScript* codes, which by itself gave +3% on MAP.
The code of the *JavaScript* parser function is the one below:

```
public String removeJavaScript(String documentBody) {
    // Get the body of the document.
    StringBuilder bodyBuilder = new StringBuilder(documentBody);

    //compiles regular expression for JS and all caps text
    Pattern jspattern=Pattern.compile("function.(.)[{]");

    int start=0;
    int end=0;
    //removes all <scripts>
    while((start=bodyBuilder.indexOf("<script", start))!=-1){
        if((end=bodyBuilder.indexOf("script>", start))!=-1){
            end=end+7;
            bodyBuilder.replace(start, end, "");
            continue;
        }
```

[2]Mean Average Precision

```
            if((end=bodyBuilder.indexOf(">", start))!=-1){
                end++;
                bodyBuilder.replace(start, end, "");
                continue;
            }
            start++;
        }

        //removes JS
        Matcher m = jspattern.matcher(bodyBuilder);
        while(m.find()){
            start=m.start();
            int count=1;
            for(int i=start; i<bodyBuilder.length(); i++){
                if(bodyBuilder.charAt(i)=='{'){
                    count++;
                    continue;
                }
                if((bodyBuilder.charAt(i)=='}')){
                    if(--count==0){
                        bodyBuilder.replace(start, i, "");
                        m = jspattern.matcher(bodyBuilder);
                        break;

                    }
                }
            }
        }

        return bodyBuilder.toString();
    }
```

The structure of the parsed document is defined in the *ParsedTextDocument* class, and it is composed of just two fields, as provided by CLEF:

1. *id*: the identifier of the document,
2. *body*: the (parsed) content of the document.

The constructor of *ParsedTextDocument* is the following:

```
    /**
     * Creates a new parsed document.
     *
     * @param id the document identifier.
     * @param body the document body.
     */
    public ParsedTextDocument(final String id, final String body)
```

Inside it, multiple controls about the validity and integrity of the parameters are performed, then an object of the class is instantiated.

The class also contains the following utility methods:

- *getIdentifier*, which returns the (unique) document identifier,
- *getBody*, which returns the body of the document,
- *toString*, which returns a *String* representation of the document,
- *hashCode*, which returns the hash code of the document identifier,
- *equals(Object o)*, which returns true if the two objects have the same identifier.

2.5. Analyzer

The Analyzer is responsible for analyzing the extracted documents and preparing them for the Indexing and Searching phases. It does so by combining a series of techniques of text processing such as tokenization, stemming, stopword removal, and many more.
We extended Apache Lucene's Analyzer abstract class [5] by creating a custom class *CloseAnalyzer*, which is fully customizable by its parameters that can be chosen when creating an instance of the class. This has been done because we tried different settings and approaches to maximize the results and kept all the possible variations as optional settings. This *CloseAnalyzer* is passed as a parameter and then used by the *DirectoryIndexer* and by the *Searcher*.
The constructor of *CloseAnalyzer* accepts the following parameters:

- **tokenizerType**: used to choose between three standard *Lucene* tokenizers: *WhitespaceTokenizer* [6], *LetterTokenizer* [7], and *StandardTokenizer* [8].
- **stemFilterType**: the possible choices for the stemming types are four standard *Apache Solr* [9] filters: *EnglishMinimalStemFilter* [10], *KStemFilter* [11], *PorterStemFilter* [12], and *FrenchLightStemFilter* [13]. We also tried using *FrenchMinimalStemFilter* [14] and a custom filter called *LovinsStemmerFilter* based off a LovinsStemmer [15] implementation but decided to keep them commented as they didn't improve the results.
- **minLength** and **maxLength**: these are integers that simply specify the minimum and maximum length of a token, applying Lucene's *LengthFilter* [16].
- **isEnglishPossessiveFilter**: specifies whether to use Lucene's *EnglishPossessiveFilter* [17] or not. Of course, this can be useful when operating with the English dataset.
- **stopFilterListName**: with this parameter, it's possible to insert the path of an eventual word stoplist *.txt* file located in the *resources* folder. To do this we use Lucene's *StopFilter* [18] and a custom class called *AnalyzerUtil* that uses a *loadStopList* method to read and load all the stoplist words from the specified file. The stoplists we created are based on the standard ones but modified after inspecting the index with the *Luke* [19] tool. We have lists of different lengths and different ones for French and English.
- **nGramFilterSize**: if specified, this parameter is used to define the size of the n-grams to be applied by Lucene's *NGramTokenFilter* [20].
- **shingleFilterSize**: similar to the previous one, if used, this integer number indicates the shingle size to be applied by Lucene's *ShingleFilter* [21] that allows the creation of a combination of words.

- **useNLPFilter**: this boolean allows the use of Solr's [9] *OpenNLPPPOSFilter* [22] for Part-Of-Speech Tagging and of a custom class called *OpenNLPNERFilter* for Named Entity Recognition. To load the *.bin* models, which are located in the *resources* folder, we use two methods from *AnalyzerUtil*: *loadPosTaggerModel* and *loadNerTaggerModel*.
- **lemmatization**: specifies whether to use Solr's *OpenNLPLemmatizerFilter* [23] by loading a *.bin* model file in the *resources* folder using AnalyzerUtil's *loadLemmatizerModel* function.
- **frenchElisionFilter**: we applied this only when using the French dataset by adding Lucene's *ElisionFilter* [24] with an array of the following characters: 'l', 'd', 's', 't', 'n', 'm'.

On top of this, a *LowerCaseFilter* [25] is always applied.
We also tried Lucene's *ASCIIFoldingFilter* [26] and *SynonymGraphFilter* [27]. For the second one, only for the French Dataset, we used a *SynonymMap* [28] based on a *.txt* file containing French synonyms.
After different trials with different variations of the parameter, the following is the instance of the *CloseAnalyzer* we used:

```
final Analyzer closeAnalyzer = new
    CloseAnalyzer(CloseAnalyzer.TokenizerType.Standard, 2, 15, false,
    "new-long-stoplist-fr.txt", CloseAnalyzer.StemFilterType.French, null,
    null, false, false, true);
```

We have opted for the French dataset and by doing so we have the *StandardTokenizer*, 2 and 15 as minimum and maximum token length, we use *frenchElisionFilter*, *FrenchLightStemFilter*, and a list of 662 French words as a stoplist. We didn't use any of the other parameters. We utilized the Gson library to efficiently parse a JSON files that contained query expansions. By leveraging Gson's capabilities, we were able to seamlessly convert the JSON data into Java objects, enabling effortless manipulation and integration of the query expansions into our application.

2.6. Indexer

The *Indexer* is in charge of calling the *Parser* (2.4) and the *Analyzer* (2.5), taking the document processed by these two and indexing them as an object of the *ParsedTextDocument* defined in the *Parser*.
The class which performs the document processing mentioned above is *DirectoryIndexer*, whose constructor is the following:

```
    public DirectoryIndexer(final Analyzer analyzer, final Similarity
        similarity, final int ramBufferSizeMB,
                        final String indexPath, final String docsPath, final
                            String extension,
                        final String charsetName, final long expectedDocs,
                        final Class<? extends DocumentParser> dpCls) {
```

It takes the following inputs:

- **analyzer**: the *Analyzer* to be used, in our case *CloseAnalyzer*;
- **similarity**: the *Similarity* function to be used by the *Analyzer*;
- **ramBufferSizeMB**: the size in megabytes of the RAM buffer for indexing the documents;
- **indexPath**: the directory where to store the index;
- **docsPath**: the directory from which the documents of the collection have to be read;
- **extension**: the extension of the files in the collection to be indexed;
- **charsetName**: the name of the charset used for encoding documents;
- **expectedDocs**: the total number of documents expected to be indexed;
- **dpCls**: the class of the *DocumentParser* to be used, in our case *ClefParser*.

The function which actually performs the storing of the index is:

`public void index()`

Its main functionality, which consists of reading, indexing, and storing the processed documents, is shown below:

```
// Create a stream of parsed documents from the file with the given Parser
    class(dpCls)
DocumentParser.create(dpCls, Files.newBufferedReader(file, cs)).forEach(pd
    -> {
    Document doc = new Document();
    ParsedTextDocument ptd = (ParsedTextDocument) pd;

    // add the document identifier
    doc.add(new StringField(ParsedTextDocument.Fields.ID,
        ptd.getIdentifier(), Field.Store.YES));

    // add the processed document content
    doc.add(new BodyField(ptd.getBody()));
    try {
        writer.addDocument(doc);
    } catch (IOException e) {
        e.printStackTrace();
    }

    docsCount++;

    // print progress every 10000 indexed documents
    if (docsCount % 10000 == 0) {
        System.out.printf("%d document(s) (%d files, %d Mbytes) indexed in
            %d seconds.%n",
        docsCount, filesCount, bytesCount / MBYTE,
        (System.currentTimeMillis() - start) / 1000);
    }
});
```

The *Indexer* contains some printouts used to see the progress of the indexing during the execution, such as the number of indexed documents, the time elapsed, if the number of documents indexed is less than expected, and some other eventual warning that can come out during this operation.
The overall workflow, starting from the documents collection and ending with having the indexed documents stored, is the following:

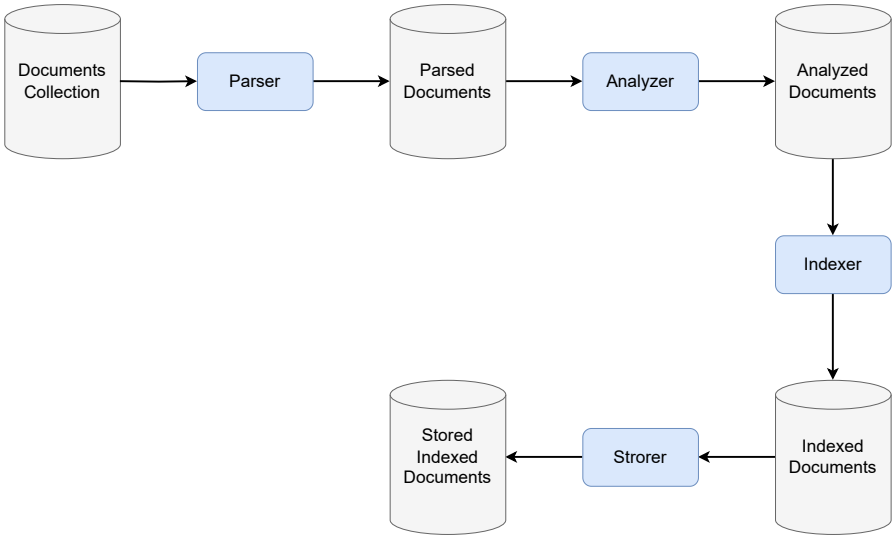

Figure 5: Workflow of the indexer.

2.7. Searcher

The purpose of the *Searcher* is to search through the indexed documents to retrieve relevant information based on user queries after analyzing them and to return a ranked list of documents that match the user's information needs.
Our implementation does so by accepting the following parameters:

- **analyzer**: in this case, an instance of *CloseAnalyzer*.
- **similarity**: we decided to opt for the *BM25Similarity* [29] function with the parameters *k1* and *b* tuned at 1.2 and 0.90.
- **Run options**: there are parameters for the index path, the topics path, the run path and the run name, the number of the expected topic (in our case 50), and the maximum number of documents retrieved (in our case 1000).
- **useEmbeddings**: a boolean value to decide whether to use embeddings or not. If it is set to true we don't use the similarity function. In our implementation, we opted to not use them.
- **reRankModel**: this is the type of model used to do a Re-Ranking on the retrieved documents. In our case, we use a model called *all-MiniLM-L6-v2* [30], explained in

the following subsection. If the parameter is set to null, no model is used and the documents are scored normally.

2.7.1. Query Expansion

When running the search function, one of the first actions performed is to generate new queries from the original ones by query expansion. We created a Python script that, given the **.trec* file, generates all the expanded terms for each query and stores everything in a *.json* file called *result*.
We use *OpenAI's Text completion* [31] endpoints to generate the expansions, we can use our need as prompt and the model will generate the result. We used the *davinci* model, which is the most powerful one, and we set the *temperature* parameter to 0.6, which is the value that gives the best results.
The sample result for prompt

"Expand the following query with num_expansions related terms or phrases for information retrieval (search-engine): query and the result should be in array format without any numbers at first [expanded_term1, expanded_term2, ...]"

is:

Query	text-davinci-002	text-davinci-003
antivirus comparaison	1. Antivirus software 2. Antivirus protection 3. Best antivirus 4. Free antivirus 5. Antivirus for Windows 6. Antivirus for Mac	1. Antivirus Reviews 2. Antivirus Software 3. Antivirus Protection 4. Malware Protection 5. Virus Scanner 6. Online Security

The main difference between *davinci-text-002* and *davinci-text-003* is that the latter has been trained on a larger dataset, allowing it to generate more accurate results [32].

2.7.2. Query Boosting

Query boosting is a technique used to assign greater relevance to certain query terms or queries.
We tried the following approach that seemed to improve the overall results: when building the queries in the search function of the *Searcher* (2.7), for each query, a *BooleanQuery* [33] is built in the following way: after getting the query expansions, each of them is added to the *BooleanQuery* with the clause *SHOULD* (meaning that at least one of them must be satisfied) and a main query is added with the clause *MUST*, indicating that it must be satisfied.
This main query is boosted using Lucene's *BoostQuery* [34], with a boost value tuned at 14.68 multiplied by the number of expansions.

2.7.3. Document Re-Ranking

This is the process of ranking the documents retrieved by the search function of the *Searcher* (2.7) a second time, using, in our case, a model that maps sentences and paragraphs to a 384-dimensional dense vector space called *all-MiniLM-L6-v2* [30]. This sentence transformer model is coming from a Python framework called *Sentence Transformers* [35], also *all-MiniLM-L6-v2* aims to train sentence embedding models on very large sentence level datasets using a self-supervised contrastive learning objective.
It works in the following way: in the constructor of the *Searcher* the Re-Ranker is initialized and a predictor is created to perform inference. At the end of the search function, we call a *sort* function that, using the predictor, creates embeddings for the documents and, for each query, calculates the similarity between the query and the documents. Finally, these scores are multiplied by the actual document's scores, and the documents are then sorted by looking at these new scores.
We tested different combinations of parameters to find the best one, and we found that the best combination is multiplying the document's score by the *BM25Similarity* function, with the similarity calculated with the *cosine* function.

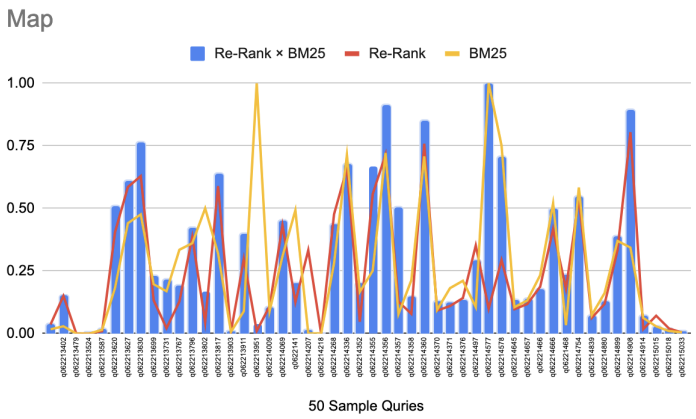

Figure 6: Plot of the re-ranking operation performed on 50 sample queries

3. Experimental Setup

3.1. Collections

We developed our model using a collection of 1,593,376 documents and 882 queries provided by *Qwant* search engine, available at https://lindat.mff.cuni.cz/repository/xmlui/handle/11234/1-5010.
The collection contains information about user web searches and actual web pages corpora. The data was originally all in French but, for both queries and documents, an English translation is provided.

3.2. Evaluation Measures

To measure the effectiveness of our IR system we used the *trec_eval* executable by testing it with the resulting runs produced by the model and its different configurations.
We tracked improvements of the following evaluation measures generated by *trec_eval*:

- **num_ret**: number of documents retrieved for a given query.
- **num_rel**: number of relevant documents for a given query.
- **num _rel_ret**: number of relevant documents retrieved for a given query.
- **map**: Mean Average Precision, a measure of the average relevance of retrieved documents across all queries. Its range is in 0-1, where 1 indicates that all relevant documents are ranked at the top of the result list.
  It's calculated as follows:
  $$MAP = \frac{1}{N}\sum_{i=1}^{N} AP_i$$
  where N is the total number of queries and AP_i is the average precision of $query_i$ and calculated as:
  $$AP = \frac{1}{RD}\sum_{k=1}^{n} P(k)r(k)$$
  with RD the number of relevant documents for the query, n the number of total documents, $P(k)$ the precision at k and $r(k)$ the relevance of the kth retrieved document (0 if not relevant, 1 otherwise).
  A high MAP score indicates that the model effectively retrieves relevant documents for a wide range of queries.
- **rprec**: R-Precision is the precision score computed at the rank corresponding to the number of relevant documents for a given query.
- **p@5** and **p@10**: Precision at 5 and at 10 is the precision computed at the top 5 and 10 retrieved documents for a given query. They are calculated as follows:
  $$P(5) = \frac{1}{5}\sum_{k=1}^{5} r(k) \qquad ; \qquad P(10) = \frac{1}{10}\sum_{k=1}^{10} r(k)$$
  with $r(k)$ the relevance of the kth document.
  These two values can be useful to understand if the system retrieves relevant documents early in the result list, making it easier for the user to find the information needed.

3.3. Git Repository

The *git* repository of the project can be found at https://bitbucket.org/upd-dei-stud-prj/seupd2223-close/src/master/ and it is organized as follows:

```
/
├── code/
│   ├── src/main/
│   │   ├── java/it/unipd/dei/se/
│   │   │   ├── analyzer
│   │   │   ├── indexer
│   │   │   ├── parser
│   │   │   ├── searcher
│   │   │   ├── utils <- ReRanker and array converter util
│   │   │   └── CloseSearchEngine.java <- main class
│   │   └── resources <- word stoplists and models for NER and POS
│   ├── python_scripts <- python scripts, results for embeddings, ...
│   └── pom.xml
├── runs <- run files
└── results <- result files
```

3.4. System Hardware

For the most time during the development of the system every member of the group ran the model on its own system, but after implementing the deep learning techniques we decided to switch and start running the tasks using GPU to improve time performances. The following are the specifics of the machines used after switching to computing also with GPU:

- CPU: Intel® Core™ i9-12900H 12th generation
- GPU: NVIDIA RTX™ A 2000 4GB GDDR6
- RAM: 16 GB SO-DIMM DDR5 4800MHz
- SSD: 512 GB M.2 2280 PCIe Gen4 TLC Opal

- CPU: Ryzen 7 1700 overclocked at 3.8GHz
- GPU: RTX 3070 TI 8GB GDDR6X
- RAM: 16GB DDR4 3000MHz
- SSD: Samsung evo 970 250GB

4. Results and Discussion

metrics	run1	run2	run3	run4	run5
num_q	657	669	669	667	667
num_ret	646525	658446	658347	652222	657903
num_rel	2550	2611	2611	2603	2600
num_rel_ret	1772	2182	2191	1866	2232
map	0.1307	0.2022	0.2335	0.1856	0.2351
gm_map	0.0117	0.046	0.061	0.0239	0.0629
Rprec	0.1041	0.1697	0.1989	0.1654	0.2022
bpref	0.3142	0.3734	0.3869	0.3466	0.3861
recip_rank	0.2436	0.3287	0.3891	0.3441	0.3945
iprec_at_recall_0.00	0.2553	0.3499	0.4134	0.3584	0.4182
iprec_at_recall_0.20	0.2387	0.3324	0.3873	0.3353	0.3927
iprec_at_recall_0.40	0.1441	0.2316	0.2732	0.2066	0.2716
iprec_at_recall_0.60	0.0965	0.1786	0.1996	0.1388	0.2014
iprec_at_recall_0.80	0.0628	0.1178	0.1311	0.0887	0.1295
iprec_at_recall_1.00	0.0525	0.0954	0.1031	0.0704	0.1028
P_10	0.0848	0.1296	0.1435	0.1126	0.1432
P_100	0.0186	0.0256	0.0268	0.0222	0.0268
P_1000	0.0027	0.0033	0.0033	0.0028	0.0033
recall_10	0.2166	0.3352	0.367	0.2849	0.3621
recall_100	0.4718	0.6426	0.6714	0.5536	0.6723
recall_1000	0.6816	0.8192	0.8218	0.7004	0.8392
infAP	0.1307	0.2022	0.2335	0.1856	0.2351
gm_bpref	0.0152	0.0387	0.0405	0.022	0.038
utility	-978.6621	-977.701	-977.5262	-972.2489	-979.6687
ndcg	0.2719	0.3655	0.3924	0.3291	0.3982
ndcg_rel	0.236	0.3119	0.3416	0.2939	0.3471
Rndcg	0.1708	0.2387	0.2657	0.2271	0.2714
ndcg_cut_5	0.1285	0.1908	0.2232	0.1854	0.2269
ndcg_cut_10	0.1609	0.2426	0.2739	0.2227	0.2758
ndcg_cut_100	0.2351	0.3349	0.3652	0.3016	0.3678
ndcg_cut_1000	0.2719	0.3655	0.3924	0.3291	0.3982
map_cut_10	0.1046	0.1665	0.1975	0.1556	0.1993
map_cut_100	0.1284	0.2	0.2315	0.1836	0.2328
map_cut_1000	0.1307	0.2022	0.2335	0.1856	0.2351

Table 1: English results from TREC eval

- **run1**: PorterStemFilter, Standard tokenizer, LenghtFilter between 2 and 15, "long-stoplist-fr.txt" for StopFilter, LowerCaseFilter, BM25Similarity;
- **run2**: FrenchLightStemFilter, Standard tokenizer, LenghtFilter between 2 and 15, "long-stoplist-fr.txt" for StopFilter, LowerCaseFilter, BM25Similarity;
- **run3**: FrenchLightStemFilter, Standard tokenizer, LenghtFilter between 2 and 15, "long-stoplist-fr.txt" for StopFilter, LowerCaseFilter, BM25Similarity((float)1.2,(float)0.90);
- **run4**: PorterStemFilter, Standard tokenizer, LenghtFilter between 2 and 15, "long-stoplist.txt" for StopFilter, LowerCaseFilter, BM25Similarity((float)1.2,(float)0.90), EnglishPossessiveFilter;
- **run5**: FrenchLightStemFilter, Standard tokenizer, LenghtFilter between 2 and 15, "new-long-stoplist-fr.txt" for StopFilter, LowerCaseFilter, BM25Similarity((float)1.2,(float)0.90), ElisionFilter with some common french articles and prepositions;

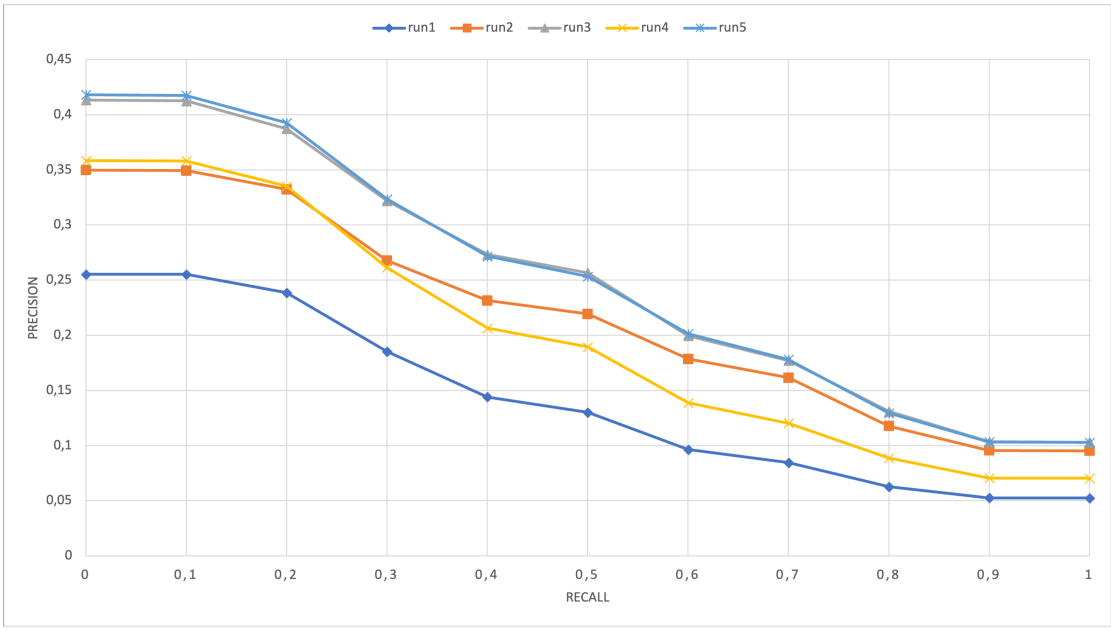

Figure 7: Recall and precision graph

In this section we provide some of the most relevant results we got during the development phase. We are considering five principal milestones that, within many different trials, led us to improve significantly our MAP score and the overall number of relevant documents actually retrieved.

First of all, given that we were provided with two different version of the same document's *corpora*, our first idea was to try with the English version.
We noticed that the best combination of basic IR tools were to use the *Porter* stemmer, a length filter from 1 to 10, and a list of stop-word composed by some standard terms

and more from the top 600 extracted from the index. The very first big milestone, that helped us to increment the MAP of around +3.5%, was the *JavaScript* code cleaner, since we noticed by inspection that many documents were having these type of scripts inside.
Always by inspecting some documents and queries, and also considering that the original collection was the French version (translated then in English), we observed that the translation was very poor: by switching to French with just cleaning the *JS* code and some other minor cleaning tools, without even using an adequate stop-list and a correct stemmer for the French language, the MAP was increasing by +5%.
2 more points were achieved with a stop list built for French in the same way we did previously for English, the *FrenchLightStemFilter* [13] as stemmer, and moving the length filter from 2 to 15 (as we noticed French tends to have longer words).
We tried some NLP[3] techniques for English, in particular using PoS[4] techniques, to see if there were improvements, and in case apply it to our main implementation for French with an appropriate model, but results were not interesting, and also the computing time was definitely too costly. Another approach we tried and that carried an improvement was to use *Query expansion*: first we used some generative text models to expand our queries, then we decided to weight different query scores by boosting the original one linearly with respect to the number of expansion used. This made us gain an extra MAP point.
We try to generate the embeddings for each document based on word2vector, we use pre-trained word2vec model *frWac_no_postag_no_phrase_500_cbow_cut100* [36] for French. Then we calculate the embedding for each document and index them as KnnFloatVectorField in lucene and use KnnFloatVectorQuery for searching the query to find the k nearest documents to the target vector according to the vectors in the given field, but the results (overall map 0.08) are not satisfying us and everything goes worser than the case of indexing and searching without anything.
We tried to combine different similarities rather than using the classic *BM25Similarity*: we tried to use the Lucene *MultiSimilarity* [37], that allows to combine equally the score of two or more similarity scores, but it does not allow to tune the weights. Then, we tried to reimplement the *MultiSimilarity* class with tuning options, but results were always lower than the standard *BM25Similarity*. Some minor improvements came up by fine tuning the document-length normalization b parameter and the term frequency component $k1$ parameter of the *BM25Similarity*.
The last main implementation we did, was to use some *Re-ranking* techniques to improve the results of the first retrieval phase. We tried to use the *SBERT* model [38], which is a pre-trained model for sentence embeddings, and we used it to calculate the similarity between the query and the document. We tried to use different distance metrics such as *CosineSimilarity* and *ManhattanDistance*, but at the end of the day, *CosineSimilarity* is much better than others.
Lastly, some minor adding were set on the *Analyzer* ([**?**]) by implementing the Lucene

[3]Natural Language Processing
[4]Part of Speech

ElisionFilter (for French) [24], which aims to remove apostrophes articles and prepositions from tokens (for example, *m'appelle* and *t'appelle* become the same token).

5. Conclusions and Future Work

In this work, we presented our approach to the CLEF *Long Eval LAB 2022* task, which aimed to develop an effective and efficient search engine for web documents.
Our approach consisted of using a combination of different techniques, including query expansion, re-ranking, and the use of large language models such as *ChatGPT* and *SBERT*.
Our experiments showed that our approach achieved good results in terms of effectiveness and efficiency, outperforming the baseline system provided by CLEF. Specifically, we found that combining two different scores in the re-ranking phase led to significant improvements in the retrieval performance. Moreover, we identified several areas for future work that could further improve the effectiveness and efficiency of our approach.
One possible direction for future work is to find better ways to combine scores or add other scores to the re-ranking phase. We plan to explore different combinations of scores and investigate the use of other large language models, such as other available *BERT* models trained, or to train some specifically for this task.
Another area for future work is to find better prompts [39] to use in *ChatGPT* for improving query expansion. We also plan to investigate the use of other LLM[5] techniques for query expansion.
We also want to explore ways to increase the similarity in *SBERT* [38], to increase the number of relevant documents found in the re-ranking phase. One possible approach is to fine-tune the *SBERT* [38] model on our specific task.
Another direction for future work is to index documents as vectors and use them directly, instead of calculating them in re-ranking. This trade-off would result in the loss of one of the scores, but it would increase the re-ranking speed.
Finally, we plan to use links inside documents to extract details that may improve the searching results. We may try to find keywords in the URL path and use them to find their domain authority and take this aspect into account in the score computation.

References

[1] C. Organizers, Longeval clef 2023 lab, https://clef-longeval.github.io/, 2023. Accessed: 2023-05-20.
[2] Qwant, About qwant, https://about.qwant.com/en/, 2023. Accessed: 2023-05-20.
[3] A. Lucene, Apache lucene, https://lucene.apache.org/, 2023. Accessed: 2023-05-20.
[4] M. D. Network, Regular expressions, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_expressions, 2023. Accessed: 2023-05-20.
[5] A. Lucene, Lucene analyzer, https://lucene.apache.org/core/8_0_0/core/org/apache/lucene/analysis/Analyzer.html, 2023. Accessed: 2023-05-20.
[6] A. Lucene, Lucene whitespacetokenizer, https://lucene.apache.org/core/7_4_0/analyzers-common/org/apache/lucene/analysis/core/WhitespaceTokenizer.html, 2023. Accessed: 2023-05-20.

[5] Large Language Model

[7] A. Lucene, Lucene lettertokenizer, https://lucene.apache.org/core/7_3_1/analyzers-common/org/apache/lucene/analysis/core/LetterTokenizer.html, 2023. Accessed: 2023-05-20.
[8] A. Lucene, Lucene standardtokenizer, https://lucene.apache.org/core/6_6_0/core/org/apache/lucene/analysis/standard/StandardTokenizer.html, 2023. Accessed: 2023-05-20.
[9] A. S. Foundation, Apache solr, https://solr.apache.org/, 2023. Accessed: 2023-05-20.
[10] A. S. Foundation, Apache solr englishminimalstemfilter, https://solr.apache.org/guide/6_6/filter-descriptions.html#FilterDescriptions-EnglishMinimalStemFilter, 2023. Accessed: 2023-05-20.
[11] A. S. Foundation, Apache solr kstemfilter, https://solr.apache.org/guide/6_6/filter-descriptions.html#FilterDescriptions-KStemFilter, 2023. Accessed: 2023-05-20.
[12] A. S. Foundation, Apache solr porterstemfilter, https://solr.apache.org/guide/6_6/filter-descriptions.html#FilterDescriptions-PorterStemFilter, 2023. Accessed: 2023-05-20.
[13] A. S. Foundation, Apache solr frenchlightstemfilter, https://solr.apache.org/guide/6_6/language-analysis.html#LanguageAnalysis-FrenchLightStemFilter, 2023. Accessed: 2023-05-20.
[14] A. S. Foundation, Apache solr frenchminimalstemfilter, https://solr.apache.org/guide/6_6/language-analysis.html#LanguageAnalysis-FrenchLightStemFilter, 2023. Accessed: 2023-05-20.
[15] A. Lucene, Lucene lovinsstemmer, https://lucene.apache.org/core/6_3_0/analyzers-common/org/tartarus/snowball/ext/LovinsStemmer.html, n.d. Accessed: 2023-05-20.
[16] A. Lucene, Lucene lengthfilter, https://lucene.apache.org/core/7_0_1/analyzers-common/org/apache/lucene/analysis/miscellaneous/LengthFilter.html, 2023. Accessed: 2023-05-20.
[17] A. Lucene, Lucene englishpossessivefilter, 2023. Accessed: 2023-05-20.
[18] A. Lucene, Lucene stopfilter, 2023. Accessed: 2023-05-20.
[19] G. Code, Luke, https://code.google.com/archive/p/luke/, 2023. Accessed: 2023-05-20.
[20] A. Lucene, Lucene ngramtokenfilter, https://lucene.apache.org/core/8_1_1/analyzers-common/org/apache/lucene/analysis/ngram/NGramTokenFilter.html, 2023. Accessed: 2023-05-20.
[21] A. Lucene, Lucene shinglefilter, https://lucene.apache.org/core/4_3_0/analyzers-common/org/apache/lucene/analysis/shingle/ShingleFilter.html, 2023. Accessed: 2023-05-20.
[22] A. S. Foundation, Apache solr opennlp part of speech filter, https://solr.apache.org/guide/7_3/language-analysis.html#opennlp-part-of-speech-filter, 2023. Accessed: 2023-05-20.
[23] A. S. Foundation, Solr opennlp lemmatizer filter, https://solr.apache.org/guide/7_3/language-analysis.html#opennlp-lemmatizer-filter, 2023. Accessed: 2023-05-20.
[24] A. Lucene, Lucene elisionfilter, https://lucene.apache.org/core/7_3_1/

analyzers-common/org/apache/lucene/analysis/util/ElisionFilter.html, 2023. Accessed: 2023-05-20.
[25] A. Lucene, Lucene lowercasefilter, https://lucene.apache.org/core/8_0_0/analyzers-common/org/apache/lucene/analysis/core/LowerCaseFilter.html, 2023. Accessed: 2023-05-20.
[26] A. Lucene, Lucene asciifoldingfilter, https://lucene.apache.org/core/4_9_0/analyzers-common/org/apache/lucene/analysis/miscellaneous/ASCIIFoldingFilter.html, 2023. Accessed: 2023-05-20.
[27] A. Lucene, Lucene synonymgraphfilter, https://lucene.apache.org/core/6_4_1/analyzers-common/org/apache/lucene/analysis/synonym/SynonymGraphFilter.html, 2023. Accessed: 2023-05-20.
[28] A. Lucene, Lucene synonymmap, https://lucene.apache.org/core/7_3_0/analyzers-common/org/apache/lucene/analysis/synonym/SynonymMap.html, 2023. Accessed: 2023-05-20.
[29] A. Lucene, Lucene bm25similarity, https://lucene.apache.org/core/7_0_1/core/org/apache/lucene/search/similarities/BM25Similarity.html, 2023. Accessed: 2023-05-20.
[30] H. Face, Hugging face 'all-minilm-l6-v2' model, https://huggingface.co/optimum/all-MiniLM-L6-v2, 2023. Accessed: 2023-05-20.
[31] OpenAI, Openai text completion api documentation, https://platform.openai.com/docs/guides/completion/introduction, n.d.. Accessed: 2023-05-20.
[32] OpenAI, How do text davinci-002 and text davinci-003 differ?, OpenAI Help Center (n.d.). Accessed: 2023-05-20.
[33] A. Lucene, Lucene booleanquery, https://lucene.apache.org/core/8_1_1/core/org/apache/lucene/search/BooleanQuery.html, 2023. Accessed: 2023-05-20.
[34] A. Lucene, Lucene boostquery, https://lucene.apache.org/core/7_3_1/core/org/apache/lucene/search/BoostQuery.html, 2023. Accessed: 2023-05-20.
[35] UKPLab, Sentence transformers, https://github.com/UKPLab/sentence-transformers, 2023. Accessed: 2023-05-20.
[36] J.-P. Fauconnier, French word embeddings, 2015. URL: http://fauconnier.github.io.
[37] A. Lucene, Lucene multisimilarity, https://lucene.apache.org/core/8_0_0/core/org/apache/lucene/search/similarities/MultiSimilarity.html, n.d. Accessed: 2023-05-20.
[38] N. Reimers, I. Gurevych, Sentence-bert: Sentence embeddings using siamese bert-networks, 2019. URL: https://arxiv.org/abs/1908.10084.
[39] S. Wang, H. Scells, B. Koopman, G. Zuccon, Can chatgpt write a good boolean query for systematic review literature search?, 2023. arXiv:2302.03495.